Open-Apple™

Releasing the power to everyone.

December 1986
Vol. 2, No. 11

ISSN 0885-4017
newstand price: $2.00
photocopy charge per page: $0.15

A concise look at Apple II RAM

Caution: The following article, like many ***Open-Apple*** *articles, starts off in an engaging non-technical style designed to lure novice Apple II users into reading it. If you get into it a ways and suddenly realize you don't have half an idea what it's talking about, go back to the February 1986* ***Open-Apple****, page 2.2, and read or reread "The Magic of Peek and Poke" for this month's assignment.*

The first Apple II came with 4K of readable-writeable RAM memory (the kind that gets erased when the lights go out). The latest model, the IIgs, can easily support more than 2,000 times that much (8,192K). This month I'll tell you the story of how the II's memory grew. By telling it I hope we can answer a few of the questions Uncle DOS has been getting about the wide variety of RAM cards available today for the II-Plus, IIe, IIc, and IIgs.

The story of Apple II RAM parallels the story of RAM chips themselves. Every few years the companies that make RAM chips have been able to quadruple the number of memory bits on a single chip. At the same time, as they have gained manufacturing experience and as the size of the market for chips has grown, they have been able to lower prices. Today, the cheapest chips available, on a per-kilobit basis, are 256K chips. They cost from 1 to 2 cents per kilobit. (Or 8 to 16 cents per kilo*byte*—the 256K RAM chips typically used in Apple IIs hold 262,144 memory *bits*; it takes eight such chips to make 256K *bytes* of memory.)

Older 64K chips currently cost 2 to 3 cents per kilobit; 16K chips cost 3 to 6 cents. Newer 1,024K (1 megabit) chips currently start at about 7 cents per kilobit. If the standard chip-price cycle holds, however, these newer chips will be cheaper (on a per-kilobit basis) than today's 256K chips within a couple of years. Next will come 4 megabit chips; by the early 1990s you should be able buy your IIgs 8 megabytes of RAM (16 4-megabit chips) for less than $200.

Meanwhile, back in 1976 when Steve Wozniak was designing the original Apple II, $200 could get you eight 4K RAM chips. Larger 16K chips were only on the horizon. Wozniak designed the Apple II so that it could accommodate three eight-chip groups of 4K chips—a total of 12K of RAM memory. However, with an eye toward the upcoming 16K chips, Wozniak put "memory configuration blocks" on the Apple II motherboard that allowed the 4K chips to be replaced with 16K chips, one eight-chip group at a time. Before long, Apple IIs had been configured into nine different memory sizes ranging from 4K to a massive 48K.

No other widely-available personal computer of the day could accommodate such a massive amount of memory. This is the primary reason *VisiCalc* was originally written for the Apple II. This same kind of RAM foresight was missing from the IIe and IIc—they were built to use only the cheapest memory chips of their day. Foresight is back in style at Apple now, however—look at the IIgs. Almost no other personal computer of the day can accommodate such a massive amount of memory—I can't wait to see what comes of that.

The language card. Massive though it seemed in 1979, 48K soon wasn't enough. Apple wanted to make Pascal available for the II, but just couldn't fit it into 48K. So the Apple Language Card—which extended system RAM by another 16K, to a full 64K bytes of memory—soon appeared on authorized Apple dealers' shelves.

We've talked about language-card memory frequently in the past (going into the most detail, including how to turn the card on and off, in July 1986, page 2.46-47). The significant thing about the language card was its use of "bank switching." The 6502 microprocessor can directly address only 64K of memory. Even in an Apple II with only 4K of RAM, a 12K portion of this 64K space is dedicated to built-in ROM memory and another 4K portion is dedicated to hardware control and to devices in slots. After you add 48K of RAM, all of the 6502's Post Office boxes have been rented.

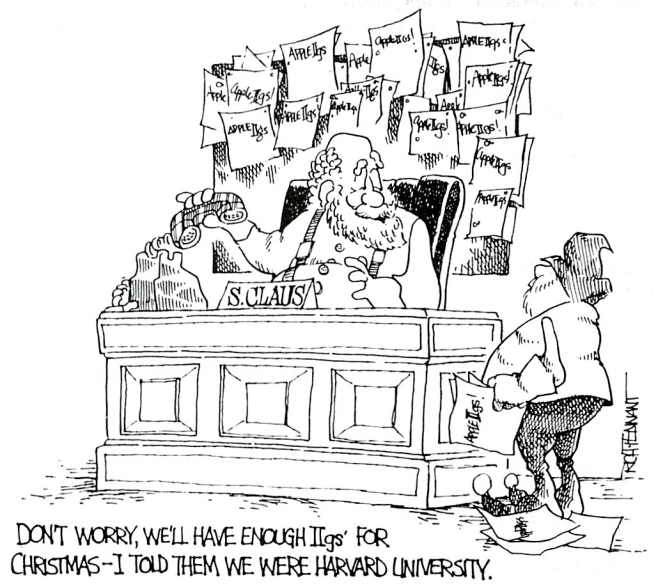
When you turn a language card on, it magically appears in the ROM's address space, from hex addresses $D000 to $FFFF. The ROM disappears. This is what bank switching is all about—electronically replacing one "bank" of memory (the Apple II's built-in ROM, in this case) with another (here, RAM on the language card).

The language card even took bank switching one step further by putting banks within a bank. The ROM address space is only big enough for 12K of RAM. To squeeze in another 4K (for a total of 16K) the language card uses the addresses from $D000 to $DFFF twice. One set of off/on switches turns the card on with the first of the two 4K banks appearing at $D000-$DFFF (and the other bank inaccessible), another set of switches turns the card on with the second 4K bank appearing at $D000-$DFFF.

After Apple lit the way with the language card, a number of third-party 16K cards appeared. (They were quite successful because Apple originally sold its card only in combination with Pascal and the combo made for an expensive package.) Then the third party manufacturers one-uped Apple by bringing out 32K, 64K, and 128K cards. Most of these cards worked by adding multiple "language cards" to the Apple.

Cards that worked like this selected the active "language card" by writing a card number (32K, 0 or 1; 64K, 0 through 3; 128K, 0 though 7) into $C084. With at least some of the cards you could also determine which "card" was active by reading $C084. Unlike Apple's 16K card, which was designed to work only in slot 0, many of the third-party cards would also work in any slot. (Add slot*16 to $C084 and to the locations given in July to manipulate cards in slots other than zero.)

An interesting variation on the theme appeared on a 256K card called the App-l-cache. Instead of being organized as multiple "language cards," App-l-cache put a "window" where everyone else put the second $D000-$DFFF bank. Any of the card's 64 4K-banks could be accessed through this window. Three of these banks also appeared at $D000-$FFFF as was normal for a



language card. The bank that was to appear in the window was selected by writing its number (0 through 63) to $C08F. By reading $C08F you could determine which bank was active. Of all the memory configuration schemes devised in the heyday of the Apple II-Plus, this one was the most elegant. Before it could take the kingdom by storm, however, it was overshadowed by the auxiliary memory scheme of the Apple IIe, which is probably the least elegant configuration ever devised.

Apple IIe auxiliary memory. For years I've been thinking that there must be a good reason for the crazy way auxiliary memory is configured on the IIe and IIc. I've been waiting for months for some software wizard to pierce the auxmem barrier and show us something of great value that can be done only because of the flip-flop, subdivided, and cross-folded format of Apple II auxiliary memory. But absolutely nothing has happened. Here, I think, is why.

The first 64K of memory on an Apple IIe or IIc is configured to look like an earlier Apple with a language card in slot 0. The second 64K of memory is configured to look just like the first—it extends from byte $0000 to $BFFF, has two banks at $D000-$DFFF, then continues in a single bank to $FFFF. This second 64K is split into two very distinct pieces—47.5K of "auxiliary" memory, and an "alternate" language card and zero page/stack. The 47.5K auxiliary memory extends from byte $200 to $BFFF. The 16.5K alternate memory extends from $0000 to $01FF and from $D000 to $FFFF.

Tools for using the second 64K of memory were built into the IIe, IIc, and later Apples. These fall into three classes—software, bank-switching softswitches, and display-switching softswitches.

The software tools include a routine called AUXMOVE that will move blocks of data between the main and auxiliary banks of the 47.5K section of memory. AUXMOVE can't access any part of the 16.5K section. To use AUXMOVE, you store the source starting address at $3C-$3D, the source ending address at $3E-$3F, and the destination address at $42-$43. You must also set or clear the microprocessor's carry bit to indicate whether you want to move from auxiliary memory to main (carry=0) or from main to auxiliary (carry=1).

A slight problem with AUXMOVE is that the routine lives in slot 3's firmware space at $C311. This is no problem on the IIc, but if someone puts a card with ROM into slot 3 on a IIe, AUXMOVE will disappear on you. (By the way, the 1985 version of the *Apple IIe Technical Reference Manual* says on page 88 that AUXMOVE lives at $C312, but don't you believe it.)

In addition to providing the ability to move *data* between banks, Apple built in the ability to transfer *control* between banks. The routine you use to do this is called XFER and lives at $C314. To use XFER, you put the address you want to jump to at $3ED-E in the bank you are in, set the microprocessor's carry bit to indicate which 47.5K bank you want switched in (0=main, 1=aux), and set the microprocessor's overflow bit to indicate which 16.5K bank you want switched in (0=main, 1=aux). Unfortunately, XFER doesn't really do all the things it needs to do if you switch 16.5K banks, but let's talk about that later.

You can always use the bank-switching softswitches, if you want, and skip the built-in software. There are four switches associated with the 47.5K section of memory. These allow you to independently select either main or aux memory for either reading or writing. If you want, you can read from one bank and simultaneously write to the other by setting these switches. They are:

```
RDMAINRAM   $C002   read main RAM
RDCARDRAM   $C003   read aux RAM
WRMAINRAM   $C004   write main RAM
WRCARDRAM   $C005   write aux RAM

To flip these switches, you must WRITE to them, not READ
```

There are only two switches associated with the 16.5K section of memory. Flipping them turns on the associated bank for both reading and writing. They are:

```
SETSTDZP    $C008   set standard zero page/stack/language card
SETALTZP    $C009   set alternate zero page/stack/language card

To flip these switches, you must WRITE to them, not READ
```

It's important to realize that the old language card softswitches have priority over these switches. If you turn on the alternate 16.5K while built-in Apple ROM appears in the language card area, all that will appear to happen will be that the zero page/stack changes. If you then turn on the language card, however, the alternate language card, not the main one, will appear in the ROM's address range.

There are also three status registers that can tell you the current memory configuration. The high bit of these registers tells you how memory is flipped (0=main, 1=aux):

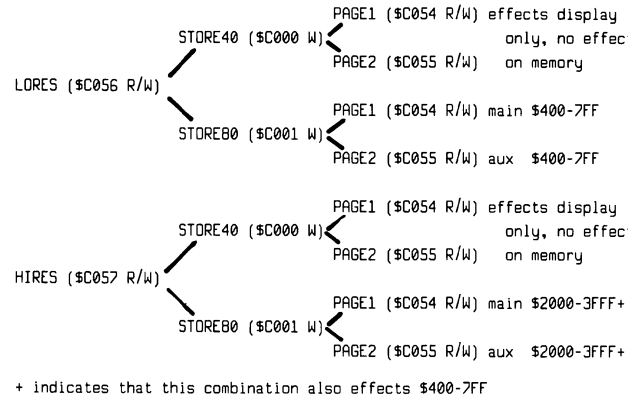
```
RDRAMRD     $C013
RDRAMWRT    $C014
RDALTZP     $C016

You must READ these status registers
```

You can also get at selected portions of auxiliary memory with the display-page softswitches. As we've discussed in the past (April 1985, pages 27-28), the 80-column screen you see on your IIe or IIc resides partially in main memory and partially in auxiliary memory (assuming the first column is "zero", the odd columns are in main memory and even columns are in auxiliary memory). In order to make it easier to access the portion of the display page that is in auxiliary memory, Apple added a softswitch that changes the function of the old PAGE1 and PAGE2 softswitches, which have been around since day one.

Normally the PAGE1 and PAGE2 softswitches flip the display between high-resolution graphics pages 1 and 2 or between text pages 1 and 2. (If you've never heard of "text page 2," don't worry, hardly anything has ever used it because it's not supported by Applesoft or the Monitor.) On the Apple IIe and later IIs, however, writing to a softswitch known as STORE80 causes the function of the PAGE1 and PAGE2 softswitches to change. After you poke STORE80, PAGE1 activates the portion of the display page that's in main memory for reading and writing, PAGE2 activates the portion in auxiliary memory.

What's more, if the computer's low-resolution/high-resolution softswitches are set for high-resolution, then PAGE1 and PAGE2, in combination with STORE80, also have an effect on the portion of memory that holds high-resolution graphics page 1—$2000-$3FFF. Thus you can actually read and write in a fairly large portion of auxiliary memory without banking in the whole thing. This effect is *in addition* to the PAGE effect on $400-$7FF area; that is, when the HIRES and STORE80 switches are on, PAGE1 and PAGE2 flip between main and auxiliary memory at *both* $400-7FF and $2000-3FFF.

Here's a summary of all this:




```
+ indicates that this combination also effects $400-7FF
R/W indicates whether the switch should be accessed with READ or WRITE
```

Interestingly, none of these switches actually do anything to the current display that appears on your monitor. There's a switch at $C050 that tells the Apple to switch to a graphics display; $C051 flips the computer back to text. There's a switch at $C00D that tells the Apple to switch to an 80-column display; $C00C flips the computer back to 40 columns. Consequently, a program can access the auxiliary memory area from $2000 to $3FFF without even affecting the screen display.

There are also status registers that can be used to determine the current status of any of these other softswitches. The high bit of these registers tells whether the feature is turned on or not (1=on, 0=off):

```
RDSTORE80   $C018   1=PAGE1/PAGE2 switches flip in main and aux memory
RDTEXT      $C01A   1=computer is displaying text, not graphics
RDPAGE2     $C01C   1=PAGE2 selected, not PAGE1
RDHIRES     $C01D   1=display is high-resolution, not low-resolution
RD80COL     $C01F   1=display is 80 columns, not 40 columns
```

Auxmem difficulties. Even with this selection of software tools, bank softswitches, and display softswitches, auxiliary memory is difficult to use. The 16.5K alternate zero page/stack/language card memory in particular is extremely difficult to work with. The nut of the difficulty is that whenever you switch in the alternate language card you also get the alternate zero-page

and stack. Things would be so much easier if these could be switched in separately.

Since they are connected you cannot, for example, use the two most common methods for passing data or parameters to a subroutine (putting the data in the stack or pointing to it with a zero-page pointer) if the subroutine you want to use is stored in or even turns on the alternate language card.

Because of this very difficulty the software tool AUXMOVE, which moves data between banks (and which uses zero page for parameter passing), can't get at either language card. XFER, which transfers control between banks, does work with the language cards (to an extent) but, as mentioned earlier, it doesn't do all it needs to do.

Remember that when you flip in the alternate 16.5K bank you get a new stack and zero page. Inside the microprocessor, however, there is a register called the stack pointer that is always aimed at the current stack position. Thus, when you flip in a new stack you also need to change the stack pointer — XFER neglects to do this. Apple's manuals put this monkey on the back of programmers. Two bytes in the auxiliary stack are to be used as storage for inactive stack pointers; $100 for the main stack pointer when the auxiliary stack is active, and $101 for the auxiliary stack pointer when the main stack is active.

Consequently, it's usually more straightforward to flip the 16.5K portion of memory with softswitches than with XFER, and to make sure the code that does the flip goes something like this:

```
1000:8D 09 C0   ALTZP   STA SETALTZP
1003:BA                 TSX
1004:8E 00 01           STX $100
1007:AE 01 01           LDX $101
100A:9A                 TXS
                        etc

1100:BA         MAINZP  TSX
1101:8E 01 01           STX $101
1104:AE 00 01           LDX $100
1107:9A                 TXS
1108:8D 08 C0           STA SETSTDZP
                        etc
```

Of course, such code must be in the 47.5K portion of memory. If it's somewhere in the language card area, flipping the switch makes the program itself disappear. It would also be wise to turn interrupts off while making the switch, and don't forget to initialize $101 in the auxiliary stack with a suitable value.

Likewise, a significant problem with using the 47.5K of auxiliary memory is that the program that flips the softswitches has to either be in both banks or it has to be in neither. When you flip the softswitch that controls which bank of memory appears in the 47.5K window while using a program that itself lies within that window, your computer crashes because the program disappears — unless, of course, the second bank holds a clone of the program in the first bank. One way around this is to use the firmware discussed earlier. The other is to move your program that flips the softswitches out of the 47.5K window — either up into the language card or down into zero page or the stack.

The zero-page/stack area is pretty precious territory to be using for a bank switching program, however, and under ProDOS the language card is where the ProDOS kernel is. However, the ProDOS development team left the rest of us a small space at $D000-D0FF in the secondary bank of both language cards to use for our own auxmem bank-switching routines.

The 64K RAMdisk. The three methods for accessing the second 64K that we've looked at so far are all built into the computer itself. ProDOS provides a fourth method of getting at this memory — it automatically sets up a RAMdisk there. Programs that do the same thing for DOS 3.3 are also available.

By far the easiest way to use the extra 64K is with the RAMdisk. The April 1986 ***Open-Apple*** discussed the possibilities, in terms of Applesoft programs, extensively.

There are several advantages to using the extra 64K as a RAMdisk (in addition to the obvious one of not having to write your own assembly language bank-switching programs). The main one is that — in addition to flipping the softswitches and moving things from bank to bank for you automatically — ProDOS *manages* the extra 64K. It keeps track of what's where. It keeps track of how much space is left. It won't accidentally overwrite something important.

The main *dis*advantage of using the extra memory as a RAMdisk is that to actually execute a program, you have to load it into the main bank of memory. If you write your own assembly language bank-switching stuff, on the other hand, you can execute programs where they are stored, without moving them from the auxiliary bank.

My opinion is that having to move routines to main memory for execution is a very small price to pay for memory management. You may disagree with me, of course — if you do, remember to disconnect the RAMdisk before you use auxiliary memory for other purposes. There is a specific protocol for this outlined in the Addison-Wesley edition of Apple's *ProDOS Technical Reference Manual* (and in Apple's "ProDOS Technical Note #8"), which I wish more software developers would follow. Software that doesn't follow this protocol (*Apple Writer*, for example), disconnects *any* storage device that appears to be in slot 3, not just ones that are using the auxiliary 64K memory bank.

Multiple auxiliary memory cards. The next chapter in the history of Apple II RAM is most interesting. An Apple engineer by the name of Peter Baum designed a memory card for the Apple II auxiliary slot that created multiple 64K banks of auxiliary memory. This card works exactly like the usual auxiliary memory scheme with one slight difference — by writing a "bank number" to a new softswitch at $C073, you can flip in an entirely new 64K of auxiliary memory.

Apple wasn't interested in the card, however, and the design ended up at a little-known Texas company called Applied Engineering. AE wrote a program that allowed AppleWorks to use the memory on the card, named the card RamWorks, and the rest is pretty much history. Other companies, notably Checkmate Technologies and Legend Industries, have since introduced similar cards and similar software, but Applied Engineering became known world-wide by dominating the market for auxslot RAM cards.

The best part of these cards has always been their support of AppleWorks. From a programmer's standpoint, they have all the bad features of standard auxiliary memory multiplied by the number of banks on the card.

In addition to the usual problems, programmers have to devise some way to keep track of which auxiliary bank is active. There is no status register on the card itself that can tell you this; you have to store the bank number in memory. Applied Engineering recommends using bytes $FFF0 in the 16.5K piece and $47B in the 47.5K piece for this. Reset doesn't automatically switch the card back to bank zero — programmers must take care to intercept Reset and do it themselves. In addition, Reset and interrupts always use some addresses they expect to find in the last few bytes of memory after $FFF0; these addresses must be written into every available bank on the card. Programs that want to support interrupts also need a special interrupt handler in each bank. Bank numbers, by the way, aren't necessarily sequential. Some cards have memory in banks 0 and 3 but not in banks 1 or 2. It depends on what kind of RAM chips (64K or 256K) were used in the card. Oh, and writing a bank number to $C073 also trips the paddle strobe, for those of you who know what that is. To ensure that the paddles are read properly, a 3 millisecond delay is required between switching banks and reading the paddles.

Since I don't recommend trying to use regular auxiliary memory for anything other than a RAMdisk, you can be sure I don't recommend using additional auxmem banks for anything else, either. Nonetheless, we've gotten several questions about how to figure out from inside a program whether an auxslot RAM card has been installed in a computer and how to figure out how much memory it has. Here's a simple Applesoft program that uses the display-page softswitches to accomplish this:

```
100 REM * Test for multiple auxiliary memory banks *

110 TEXT : HOME : VTAB 10
120 PRINT "Just a minute here...." : PRINT
130 DIM B(127) : REM array to remember which banks have memory

140 POKE 49239,0 : REM turn on HIRES ($C057)
150 POKE 49153,0 : REM turn on STORE80 ($C001)
160 POKE 49237,0 : REM turn on PAGE2 ($C055)

200 FOR BANK=127 TO 0 STEP-1
210 POKE 49267,BANK : REM $C073
220 B(BANK)=PEEK(8192) : REM save value now at $2000
230 POKE 8192,BANK : REM put bank number at $2000
240 NEXT

300 FOR BANK=0 TO 127
310 POKE 49267,BANK : REM $C073
320 IF PEEK(8192) < > BANK THEN 360 : REM If < > then no RAM bank here
330 POKE 8192,0 : IF PEEK(8192) < > 0 THEN 360 : REM double-check
340 POKE 8192,255 : IF PEEK(8192) < > 255 THEN 360 : REM triple-check
350 POKE 8192,B(BANK) : B(BANK)=1 : B=B+1 : GOTO 370
```

Ask (or tell) Uncle DOS

Some corrections and amplifications

Grab a pencil and your binder of ***Open-Apple*** *back issues and let's fix a few things subscribers have pointed out to us this month.*

November 1986. *The page between 2.74 and 2.76 is 2.75, of course, not 2.78. The page between 2.77 and 2.79 is really 2.78, not 2.75. Our index will use the correct, rather than the printed, page numbers, so change them now.*

We've gotten several requests for an AppleWorks 2.0 update to Alan Bird's "don't pass go" program for AppleWorks. The program appears on page 2.75 (you've changed the page number already, right?). It patches AppleWorks so that it doesn't stop twice and wait for keypresses on the way to the desktop. So we called Alan and found out that with version 2.0, the correct value for A1 is 14468 and for A2 is 14148.

In the software-we-forgot-about department, Alan reports (in response to a question on page 2.80) that there is a machine language sort program called QSORT.EXTRA.VAR that works with ***Extra K*** *on newer versions of Beagle Bros* ***Pro-Byter*** *disk.*

David Szetela at ***Nibble*** *reports (in response to the question about sorting long DOS 3.3 catalogs on page 2.78) that the program DISK MASTER on Nibble's* ***Disk Customizer*** *disk has been doing that for four years already ($29.95 from Nibble, 45 Winthrop St, Concord, MA 01742 617-371-1660).* ***Nibble*** *has an extensive collection of programs—take a look at the listing in the back of any issue.*

And George Tylutki reports (in response to the question about sorting long ProDOS catalogs on page 2.80) that his program on the Nite Owl ***Developer Disk #2*** *($39.95 from Nite Owl Productions, 5734 Lamar Ave, Mission, KS 66202 913-362-9898) can alphabetize up to 623 files in a single ProDOS subdirectory (if there are more, it just leaves them alone).*

Finally, last month's article about the bug in the ProDOS floppy disk driver clearly stated that it referred to ProDOS 1.1.1. No sooner had it gone out than we got a call asking "which version of ProDOS 1.1.1?" It was news to us, but it seems there are several versions, which can be identified by the "modification date" when the file is cataloged. Apple's official ProDOS 1.1.1 release has a modification date of "09/18/84." This is the version the article refers to. If you have a version with an earlier modification date, get rid of it. If you have one with a later modification date, someone, probably you, has already done some kind of modification to the file.

May 1986. *For example, you might have made the quit code modification that causes BYE to reboot, which was published in the May 1986* ***Open-Apple*** *at the beginning of page 2.31. Obviously you didn't, however, or you would have called months ago to tell us it doesn't work, as two subscribers did this month. First, there are no fewer than three mistakes in the third line of the instructions. That line should read:*

```
572D:CE F4 03 6C FC FF
```

In addition, the BSAVE command has the wrong number after the L parameter. To correct this, just scratch the L parameter out of the instructions. The paragraph that follows the instructions should say that the machine code translates as DEC $3F4 followed by JMP ($FFFC).

October 1986. *In the chart of Apple II Family Identification Bytes (page 2.66), the Apple IIe/original entry under $FBBF should be $C1, not $00.*

Control-D(efeated)

I am experiencing a problem with loading custom characters into my Imagewriter using my Apple IIc and hoped you could help. There are two methods I have used to load custom characters. The first involves the use of the PRINT statement with the CHR$(n) function to send data bytes. For example, the following produces a lower case italics "n":

```
100 PRINT "nK"; CHR$(0); CHR$(100); CHR$(28);
      CHR$(4); CHR$(100); CHR$(24); CHR$(0);
      CHR$(0);
```

While this works fine it is a time consuming method and subject to many typing errors.

Another method uses a FOR-NEXT loop and READ-

```
360 B(BANK)=0
370 NEXT
380 POKE 49267,0 : REM return $C073 to bank 0

400 PRINT "This machine has ";B;" banks of auxiliary memory, ";
410 PRINT "for a total of ";B*64+64;"K."
420 PRINT
430 PRINT "This memory appears as banks:";
440 FOR BANK=0 TO 127: IF B(BANK)=1 THEN PRINT SPC(3); BANK;
450 NEXT

460 PRINT : PRINT : END
```

The Apple memory standard. In September 1985 Apple itself introduced a standard-slot-based extended memory card for the Apple II-Plus and IIe. In September of this year Apple introduced a new revision of the Apple IIc that can also accommodate a special version of this type of card. The card for the Apple II-Plus/IIe also works in the new IIgs. In addition, memory added to a IIgs by means of *its* special memory expansion slot can be configured so that software running in a IIgs sees the memory as an Apple memory card in a standard slot.

Unlike the Apple memory card, auxslot RAM cards don't work on the II-Plus or the IIgs, which don't have an auxiliary slot. Special cards that work like auxslot cards are available for the IIc, but, nowadays, so is the Apple card. Because the auxslot RAM cards can't be used with a IIgs as anything other than a cheap source of memory chips, I recommend that you think long and hard before buying one. If you upgrade your IIe to a IIgs during the next 18 months (won't everyone?) the auxslot RAM card will be useless.

There are several significant differences between the Apple memory standard and the auxslot RAM cards. The Apple memory card was designed from the beginning to be used as a RAMdisk. The card has machine language programs built into it that automatically activate the RAMdisk feature for both DOS 3.3 and ProDOS. All the auxslot RAM cards we've seen come with software that will turn them into RAMdisks, but this software is packaged on disk and has to be run separately to activate the RAMdisk feature.

It isn't possible to execute programs stored on an Apple memory card. The memory simply isn't connected to the microprocessor. To execute a program stored on the card, the program must be loaded into main memory. As we have seen, however, for all practical purposes this is no different from what must be done with an auxslot RAM card.

Cards that use the Apple memory expansion standard are available from several suppliers other than Apple itself. Applied Engineering's version, called RamFactor, includes AE's AppleWorks expansion software, as well as the ability to partition the card into several "disks" and to boot from it. A battery-back up option is available that essentially turns the card into a small but speedy "hard drive." Cirtech's Flipper (known as Flipster in the U.S.), comes with most everything the RamFactor has except the battery backup and supports more operating systems (including all versions of Apple Pascal).

As mentioned earlier, the IIgs has a new type of memory expansion slot that can hold up to 8 megabytes of RAM and 1 megabyte of ROM. When the IIgs is operating in "IIe mode," the only reasonable way to use the extended RAM is as a RAMdisk. The RAM appears to be part of a standard Apple memory card. When in "IIgs" mode, on the other hand, the memory on this card is "linear" (not "bank switched" as with IIe auxiliary and language card memory) and is directly addressable by the microprocessor. This means the memory bytes appear in sequential memory addresses starting with byte $000000 and going up to the number of bytes of memory you have. (Eight megabytes of RAM would take you to byte $7FFFFF.) The IIgs includes a memory management tool that allocates the available RAM among programs—the built-in RAM disk is one of the programs that's likely to ask for memory.

Rules for living together in /RAM. Since the easiest way to use extended memory on a IIe or IIc is as a RAMdisk, and since programs that use IIe/IIc auxiliary memory also work on the II-Plus and IIgs RAMdisks, it seems reasonable that programmers should write software that takes advantage of RAMdisks rather than directly addressing additional memory.

Because the operating system will handle RAMdisk memory contention, it should be possible for different programs to coexist peacefully. One big problem with the auxslot RAM cards has been programs that go out to see how much RAM is available on the card and then take *all* of it, ignoring any RAMdisk that might pre-exist on the card. Instead, software authors should make their programs configurable as to how much RAM the user wants them to access. The software should then obtain that much by creating a file on the RAMdisk. If that much space isn't available, the user should be asked to delete some files from the RAMdisk. Additional RAM on the disk should be left free for other programs. In addition, programs should remember to delete their RAMdisk file as part of their quit routine.

DATA statements. For example, the same "n" can be produced by:

```
100 Z = 10
110 FOR X = 1 TO Z : READ BYTE
120 PRINT CHR$(BYTE); : NEXT
130 DATA 110,72,0,100,28,4,100,24,0,0
```

Both methods work using DOS 3.3. But using ProDOS and the second example, the Imagewriter reads the data byte "4" as control-D and interprets it as the signal to end the loading sequence. My solution has been to use the READ-DATA method for all characters that do not contain a "4" and use the PRINT statement for those characters that do.

Why does this happen? Why only with ProDOS and why only with the READ-DATA method? Any guidance would be helpful and much appreciated.

Roger H. Brown
Chesterfield, Mo.

Your supposition that the Imagewriter is interpreting the control-D as an end-of-transmission signal is incorrect. It's Basic.system that is interpreting the control-D as a signal. It thinks you want to send a DOS command. It swallows the control-D and the characters dragging their tails behind it and usually even has the nerve to call its nasty behavior ***your*** *SYNTAX ERROR (since it can't make sense of the "command").*

DOS 3.3 will do the same thing under slightly different circumstances. In both situations the specific problem is a result of a widespread, poorly understood flaw that is characteristic of using control-codes and Escape sequences to send commands to devices such as disk drives, interface cards, modems, and printers.

Control-codes and Escape sequences are here to stay, however, so it's probably best that we investigate this problem in some detail as we answer your specific question.

First, let's make clear what's meant by "using control-codes and Escape sequences to control devices." Take a printer. Usually you send it ASCII-encoded letters, numbers, and punctuation marks. It prints them. You also send it control-characters that aren't printed, but that tell the printer to do such things as returning to the left margin, underlining, and tabbing. These characters are embedded in the data stream you send to the printer along with all the letters, numbers, and punctuation marks.

Sometimes it's necessary to send non-ASCII data to your printer. This might include the bytes that make up a custom character set, as in your example, or the bytes that tell the printer how to print a copy of the Neanderthal on your high-resolution graphics screen. The ASCII meanings of this non-ASCII data will invariably include control codes, letters, numbers, and punctuation marks. Your printer doesn't print or respond to any of this, however. It knows you are sending non-ASCII data because you've told it so with ASCII codes that say "here come some custom characters" or "here comes a graphic."

However, there are at least two other creatures inspecting and sometimes manipulating the characters you are trying to get to your printer from PRINT CHR$. These idiots don't know you are sending non-ASCII data. They continue to search for, and respond to, what they think are ASCII command-codes embedded in your data stream. These folks are your printer interface card and DOS 3.3 or Basic.system.

Your printer interface card watches the data you are sending to the printer very carefully. It's searching for a control-I character, which is supposed to mean that what follows is an interface-card command. When it sees a control-I, it eats it and at least one additional character. Then the card tries to make sense of its supper. If it can, it will take some action. If it can't, your characters are simply digested and never reach the printer.

Likewise, DOS 3.3 watches all the characters you print and pounces when it sees the two-character sequence control-M control-D (a control-M is a carriage return). In your examples, if you change the 28s to 13s you'll find ***neither*** *program will work with DOS 3.3.*

Because of all the problems programmers had with the DOS 3.3 command scheme, Apple's programmers changed things slightly under Basic.system. Instead of watching for a Return, Basic.system secretly turns on Applesoft's TRACE mode and uses it to track the execution of Applesoft programs, statement by statement. Each time a new statement is executed, Basic.system looks to see if it's PRINT. If it is, Basic.system further examines the statement to see if the first character that is to be printed will be a control-D. If it is, Basic.system assumes that what follows, up to the next Return character, is a DOS command. Note that in your first example the control-D is not the first character after the PRINT statement—that program works fine under Basic.system. In your troublesome second example, however, all the characters end up being "the first character after a PRINT," including our friend CHR$(4).

So we've identified your problem—but wait, there's more. Applesoft itself always "sets the high bit" on characters it prints. Thus, although you think you've sent 0, 100, 28, 4 and so on, what your printer actually has been receiving is bytes equal to those values plus 128—128, 228, 156, 132 and so on. You didn't notice a problem, however, because the Imagewriter automatically ignores the high bit unless you tell it not to (either by sending a command or setting a dip switch). Other people have encountered this problem, however (see "A bit too many" in the April 1986 ***Open-Apple****, page 2.24).*

To solve this general category of problems once and for all, we need to figure out some way to either bypass Applesoft, DOS, and interface cards, or to tell them we are sending non-ASCII data.

In the April article just mentioned I included a short machine language routine for bypassing Applesoft. It went like this:

```
0300: A9 00     LDA #$00    load character
0302: 20 ED FD  JSR $FDED   send it to COUT
0305: 60        RTS         back to Applesoft
```

The article shows how to poke this routine into memory. To use the routine, using your second example, for example, you would change line 120 to:

```
120 POKE 769,BYTE : CALL 768
```

$FDED is the address of a routine in the Apple Monitor, known as COUT (say "see-out"), that sends the character you want to print directly to the "current output device." You may be surprised to learn, however, that both DOS 3.3 and Basic.system grab control of the critical current input- and output-device hooks and always appear to be the "current device." They do this so they can spy on all the characters you print or type. By changing the JSR $FDED into a JSR directly to the interface card, however, you can bypass DOS as well as Applesoft. That will solve your control-D problem.

Then the only problem left is deciding where to jump to on the interface card. All Apple II-compatible printer interface cards have machine language programs built into them that show up in the computer's memory at byte $Cs00, where "s" is the number of the slot the card is in. Byte $Cs00 itself on all cards is known as the Basic entry point. After you do a PR# or IN# command, this is where control passes to print or get the next character.

However, most cards use $Cs00 only as an ***initialization*** *entry point. Only the* ***first*** *call to the card is supposed to use this address (although apparently even some commercial software sends all characters here). Sending subsequent characters to $Cs00 causes the card to be reinitialized with each character. According to the IIgs documentation, "This will currently work (to a degree) on the IIgs, but applications that do this are living on borrowed time, since it is almost certain that future firmware will not permit this practice."*

As part of their initialization sequence, all interface cards I know of tell the Monitor where further characters should be sent (or where further characters can be input from) by changing the current-device hooks. Consequently, there are no standard locations for these calls. Actually monitoring the current-device hooks from inside an Applesoft program is difficult to impossible—DOS replaces the card's addresses with its own in a matter of microseconds.

The only easy way to proceed from here is to dig the addresses we need out of DOS itself—both DOS 3.3 and Basic.system squirrel the card's addresses away where they can be used when needed. DOS 3.3, when at it's standard 48K location, stores the values we need at bytes 43603-4 ($AA53-4) for output (43605-6 or $AA55-6 for input). Basic.system stores them at 48688-9 ($BE30-1) for output (43607-8 or $BE32-33 for input). We'll show how to retrieve these values in a moment.

Incidentally, many firmware cards also support a protocol known as "Pascal 1.1" that does have identifiable locations for initialization, read, write, status, and, on the IIgs, control calls. You can use these entry points from any language—they were originally developed for Pascal but are now the entry points of choice for most languages. They are, however, more difficult to use from Applesoft than the Basic entry points, because the microprocessor's registers have to be initialized to specific values before a call and because of some other important considerations—see ***Open-Apple*** *June 1986, pages 2.34-35 for a little more on this.*

While it is relatively easy to bypass Applesoft and DOS, it is much more difficult to bypass an interface card. It can be done, however, by directly manipulating the interface card's soft switches and status registers—the card's "hardware." This approach has the built-in advantage of also automatically bypassing Applesoft and DOS. It is commonly used by commercial graphic printing programs, custom font downloaders, and software that works with modems.

What makes this approach difficult is that there is no uniformity among interface cards at the soft-switch and status register level. I can give you a short subroutine that will solve the high bit and control-D problems of Applesoft and DOS on any Apple II, but giving you subroutines to directly access the hardware on a multitude of interface cards is much more complicated.

Nonetheless, that's what I've been intending to do (as you know, we've had your letter for several months, already). But I wanted to wait until the IIgs came out so that we could include subroutines to directly access its hardware, since that aspect of the

IIgs is very different from earlier Apple IIs.

Now that I've seen the serial port firmware, soft-switches, and status registers on the IIgs, however, I've decided this is the wrong approach. The IIgs itself has serial port firmware that should permanently obviate any programmer's need to directly touch the hardware. In addition, using the firmware enhances your program's compatibility in the long run, since the firmware trick that gets non-ASCII data through the serial ports on the IIgs will work on any future Apple II. In fact, the trick works on the IIc and on II-Pluses and IIes equipped with Apple's Super Serial Card and with many other (but not all) interface cards as well.

The trick is to simply include the interface card command "control-I Z" in the output stream. The Super Serial Card, the IIc, and the IIgs all recognize this as a "zap" command. After the zap command, interface card command characters aren't acted upon, but instead are sent down the data stream with everything else. This continues until the card is reinitialized with a PR# command or the equivalent. (Note that you have to resend the control-I Z(ap) every time you do a PR#1.)

So we've distilled the solution to the "embedded command character" problem to:

```
100 REM * Initialize and "zap" serial port or card *
101 REM *   You must give SLOT the proper value     *
102 REM *   before calling this subroutine.         *

110 PRINT CHR$(4);"PR#";SLOT
120 REM last chance to print any other control-I
        commands you need
130 PRINT CHR$(9);"Z"
140 POKE 768,169 : REM "LDA #"
150 POKE 770,32  : REM "JSR"

160 A=43603 : REM DOS 3.3
  or
160 A=48688 : REM Basic.system

165 POKE 771,PEEK(A)
170 POKE 772,PEEK(A+1)
180 POKE 773,96  : REM "RTS"
190 RETURN

200 REM * Send data byte, avoiding Applesoft/DOS *
201 REM *    Data byte must be placed in D       *
202 REM *    before calling.                     *

210 POKE 769,D : CALL 768 : RETURN
```

There is one other device we should mention that responds to embedded ASCII commands. That is the stand-alone modem. When on-line, Apple's Personal Modem and other "Hayes-compatible" products look for the character string "+++" and respond to it. What Apple's modem manual doesn't make at all clear is that it also looks for a one-second delay before and after the three plus signs. Without those delays the plus signs are not recognized as the beginning of a command code. If you've had trouble getting macros to work with your modem, this is probably why. Simply add one-second delays before and after the plus signs in your macros and they should begin to work.

Which revision A?

The manual that came with my IIe says I must have a revision B or later motherboard in order to use double high-res graphics. But a knowledgeable friend says if an "enhanced" sticker was on my IIe, I can use double high-res. What's the straight dope? Also, is there any software for the IIe that uses double high-res?

David A. Bixler
St. Louis, MO

Apple's inability to count past "B" on its IIe motherboard designations has confused lots of people. There are actually two vintages of IIe motherboards that have an "A" following their serial number.

The original revision A motherboard, which was not able to handle double high-res graphics, was confined to Apple IIes built in early 1983; any IIe you buy now should be capable of double high-res unless you buy a very old used one that was never upgraded. The new revision A board, which can handle double high-res just fine, can be easily identified—most of the chips are soldered directly to the board rather than being placed in sockets.

The presence of an "enhanced" sticker doesn't necessarily mean you can do double high-res—a 1983-vintage revision A motherboard can be "enhanced" and the Apple IIe Enhancement Kit includes the sticker. Enhanced IIes that can't do double high-res are probably very rare, however.

Double high-res graphics software is indeed available; popular examples are ***Beagle Graphics*** *from you-know-who and* ***Dazzle Draw*** *from Broderbund Software.*

IIgs alternate display mode

I went down to my local computer store with a couple of my own disks and used the new Apple IIgs. Guess what? It runs APPLEVISION without a hitch! (See ***Open-Apple***, July 1985, page 50.) Granted, it worked better in the "normal" mode (in "fast" the song sounded like a 33 record played at 45 rpm), but it still ran. One thing puzzles me, though: what is the "alternate display mode" that shows up on the IIgs control panel? I asked the salespeople at two stores, and no one seems to know.

Eric Patterson
Medford, OR

"Alternate Display Mode" is one of two ROM-based "desk accessory" programs on the IIgs; the other is the control panel itself. Selecting the alternate display mode allows the IIgs to display data from text page 2, which is unsupported by the Apple IIgs hardware.

So you can understand why, I'll have to explain a little about how the IIgs works. We've talked several times in the past about the Apple II's "memory-mapped I/O." This means that all the data that moves into or out of the computer passes through what appears to the microprocessor to be standard memory cells. For example, in the standard Apple II, what appears on your monitor's text display is a reflection of data held in the memory area from $400 to $7FF. When you move the cursor across the AppleWorks screen, what is really happening inside your computer is that the applications software and operating system software are working together to change the values in this memory area so that what appears to be a cursor will appear to move.

The Apple II video generation hardware accesses memory in lock step with the microprocessor. Each gets its turn once every millionth of a second. Consequently, the timing of the video generation hardware and of the microprocessor are tied together.

When the IIgs was designed, the engineers wanted to let the microprocessor run faster while allowing the video generation hardware to continue to run at the old speed. To accomplish this, they split up the 256K of RAM that's built into the IIgs into "fast RAM" and "slow RAM." The slow RAM is the built-in RAM in banks $E0 and $E1. The fast RAM is in banks $00 through $7F (only $00 and $01 are built-in).

Before anything can appear on the IIgs screen, it must be stored in the proper place somewhere in bank $E0 or $E1. However, programs written before the IIgs appeared don't know anything about banks $E0 and $E1. They are loaded into and run in banks $00 and $01. In order to get the data that APPLEVISION and AppleWorks store in banks $00 and $01 to appear on the screen, Apple's engineers gave the IIgs the power to "shadow" anything written into the display-page areas of banks $00 and $01 into the equivalent position in banks $E0 and $E1.

So, programs run in fast RAM while the video circuitry works like it always did in slow RAM. The IIgs hardware automatically takes care of moving stuff from fast RAM to slow RAM for older non-IIgs software, while software written specifically for the IIgs will turn shadowing off, use bank $00 as prime real estate rather than as graphics pages, and use banks $E0 and $E1 for all I/O. However, when accessing banks $E0 and $E1, the IIgs microprocessor has to slow down to match the pace of the video circuitry.

The only hitch with all this is that the engineers didn't include "text page 2" in the shadowing scheme because they didn't think it had been used enough to become a compatibility issue. Text page 2 is the memory area from $800 to $BFF. It has always been available in terms of the Apple hardware, but Apple II operating systems have never supported it. After the hardware design of the IIgs display modes had been locked in, however, several significant Integer Basic Apple II programs that used the page 2 display were found to be incompatible with the IIgs.

To make them compatible, Apple's software engineers added "alternate display mode" as a desk accessory. When you turn on alternate display mode, it hooks itself into the IIgs "heartbeat interrupt." At each heartbeat, alternate display mode quickly copies everything in the text page 2 area in bank $00 to bank $E0. This solution allowed Apple to transparently support text page 2 without having to re-design the IIgs hardware. For an comparison of the two modes, run the old Integer Basic program THE INFINITE NUMBER OF MONKEYS with and without alternate display mode enabled.

RamFactor as hard disk

You've mentioned Applied Engineering's RamFactor card several times but never in much detail. The possibilities this card offered were too great to pass up. I got one with 1 megabyte of RAM and the battery backup to preserve the card's memory when the computer is off. The battery backup, by the way, was back ordered for the better part of 2 months. I thought you and your readers might like some first-hand impressions of this device. In a word—it's fantastic!

The reviews of it in magazines have mostly emphasized it as a memory card for use with AppleWorks on an Apple II-Plus. But RamFactor is more than that—in fact, I have a IIe that already had a RamWorks card installed. By putting RamFactor in slot 7 it became the device that boots when I turn my computer on. I have configured it as a RAMdisk containing ProDOS, *Apple Writer*, *SuperCalc*, *Filing System*, and various utilities. It boots instantly, then one keypress gets me into any application within one or two seconds—no waiting, no searching for the right disk. Changing applications is just as fast (open-apple/control/reset plus one keypress): from *SuperCalc* to *Apple Writer* takes three seconds total.

The battery backup normally provides power to the

card from house current but switches to battery should house current fail. This feature works, by the way; we lost power for a few minutes a week ago and the battery preserved every bit on the card. Should the card be erased for some reason, it can be reformatted and loaded in less than half an hour with help from a good file copier (I use both FILE.MOVER from Beagle Bros *Big U* and *Copy II Plus*). I don't use the card to store valuable data but this isn't a drawback for someone who has lots of little files on floppies rather than a few big ones. In fact, the only drawback to this great convenience is the price tag—$440 for the whole setup, cheaper than a hard disk but not cheap.

Robert H. Holdsworth
Wilbraham, Mass.

I've been using one of Apple's memory cards just as you describe for several weeks. I'd rather have a RamFactor—Apple's card won't boot, doesn't have a battery backup option (I just leave that computer on all the time—it's connected to a $250 uninterruptable power supply), and can't be partitioned for various operating systems. I used the BACKUP program on Glen Bredon's ***ProSel*** *package to save the contents of the RAMdisk onto 3.5 inch disks after I loaded it the first time. When I have to turn the computer off for some reason I can reload the RAMdisk with Bredon's RESTORE in less than 3 minutes.*

A buffer in headache

Here is one that has had me climbing the walls for days. Enclosed is a program was intended to show how to direct output to the printer and return to the screen. Imagine my dismay when it didn't work!

As shown, the program will only activate the printer when the stars are aligned correctly or some other esoteric criteria are met. When hard-copy is requested, you probably won't get it.

It works just fine, however, if you change line 190 from:

```
190 PRINT X(Q);" ";
```

to

```
190 PRINT X(Q)
```

Using the Applesoft TRACE command even shows the line is being executed, but it doesn't activate the printer. (Well, usually doesn't activate the printer.)

Obviously, this isn't how I intended it to work. There is no major problem with the program printing the values on new lines rather than the same one, but I just don't understand why it does (or doesn't) do what it does or doesn't do.

Dwayne C. Smith
Noadron, Let.

This one stumped me, but Dennis finally figured it out. The problem sequence is:

```
180 PRINT D$;"PR#1"
185 FOR Q = 1 to A
190 PRINT X(Q);" ";
200 NEXT Q
205 PRINT D$;"PR#0"
```

As you know, the semicolon at the end of line 190 suppresses a carriage return each time through the loop. As it turns out, however, you ***never*** *send a carriage return to the printer within your loop; then you terminate the printout with PR#0.*

Most of today's printers contain at least a small internal RAM buffer that receives characters and stores them. This is so that the printer can receive characters at a constant rate but print them in spurts (one line at a time, followed by a delay while the print head repositions itself). We suspect your printer is set up so that the contents of the buffer aren't printed until a carriage return is received; this is a common configuration.

If you send less than a buffer-full of data to a printer without using a carriage return, nothing will ***appear*** *to happen. But the data will be in there, ready to spoil the page of the next person who sits down at the computer and tries to print something—or to really fool you by printing out if you run the program a second time (the return at the end of PR#1 will trigger it). With some printers, including Apple's, pressing the select switch will also cause the contents of the buffer to be printed.*

Add the following line to your program:

```
203 PRINT
```

This will terminate the print out with a hard carriage return. If that doesn't solve your problem, let us know and we'll give you ***Nibble's*** *phone number.*

Kudos of the month

I write to you on behalf of CondiCom and your many readers who use *Apple Writer* and are not aware of CondiCom's gem of a utility called *OpenAppleWriter*, which was mentioned in a letter in ***Open-Apple*** last September (page 71). Instant *Apple Writer* on the Sider is a joy to behold. The gentleman at CondiCom had the uncommon courtesy to find my telephone number and call long distance to verify my particular hardware before shipping my order. Within half an hour of it's arrival, *Apple Writer* was enhanced and tucked away deep inside my Sider. The documentation is like ***Open-Apple***, sparse but potent.

Peter Walmsley
Fort Lauderdale, Fla.

OpenAppleWriter *is $39 for the DOS 3.3 version, $29 for the ProDOS version, $49 for both from CondiCom, 436 Berry Drive, Naperville, IL 60540 312-357-0274.*

RAM Van Lines

How can I load large (>128K) programs into RAM disk automatically upon start-up?

Marc Odin
Minneapolis, Minn.

It depends on which operating system you are using.

For ProDOS, a BASIC.COPY program was published in ***Open-Apple*** *in July 1985 (pages 50-52), with an important addition and correction in the October 1985 issue (pages 74-76). Its whole reason-for-being was to demonstrate how to easily copy very large files. Most RAM cards come with a similar program—look on the utility disk that came with your card—however, some of these may not handle large files.*

Another ProDOS alternative—the one we use around here—is to use the disk backup and restore utilities in Glen Bredon's ***ProSel*** *package ($40, 521 State Road, Princeton, NJ 08540). They allow you to backup the contents of a RAMdisk into a* ***file****, which can then be kept on one of your disks. The RAMdisk can be automatically restored from that file at startup, or you can make restoration a manual procedure.*

For DOS 3.3 the easiest way to proceed is to use an EXEC file with FID. The following EXEC file, for example, will copy all the files on the boot disk in slot 6 to a RAM disk in slot 4, drive 1:

```
BRUN FID
1
6
1
4
1
=
N
XX9
```

Your boot disk must include both the EXEC file and FID itself. If you name the EXEC file COPY TO RAM and you have room for a short HELLO program, the following will get all this to work automatically:

```
10 REM you may need to BRUN a program to turn
20 REM   your memory card into a RAMdisk here
30 PRINT CHR$(4);"EXEC COPY TO RAM"
40 END
```

The EXEC file BRUNs FID, enters a "1" when FID's function menu appears (the "COPY FILES" option), enters slot 6, drive 1 for the source drive and slot 4, drive 1 for the target. The next prompt is for the filename; we enter an equals sign to indicate we want all files, followed by a "N" to indicate we do not want prompting after each filename. The two "X"s in "XX9" start and end the copy (each "X" responding to FID's request for a keypress to continue). The "9" takes us out of FID via menu option 9.

For either DOS or ProDOS you might want to add commands to check for the RAM card and see if it already contains the files before you copy them. This would happen, for example, if you had to reboot your computer but hadn't turned it off.

Dennis says that for CP/M there are SUBMIT and XSUB commands that can be used to issue commands similar to the EXEC command. SUBMIT allows you to enter CP/M system commands, while XSUB extends the ability to allow input into programs themselves. You can use these in conjunction with PIP to copy files. For example, a SUBMIT file containing the command PIP C:=A:.** would copy all files from drive A: to C:, using the "*.*" wildcard to represent all filenames. Some CP/M versions allow you to install a file to AUTORUN after CP/M boots; if your system has this feature and you can install the SUBMIT file you can automate the whole procedure.*

If there's an easy way to load large files into a RAMdisk automatically from Apple Pascal, we don't know what it is.

NTSC and PAL Apples

I would like to know what are the differences between an American (NTSC) Apple IIe and a European (PAL) Apple. Could you please list out the differences?

Tai Fan Li
Kuala Lumpur, Malaysia

Jim Sather has described the circuitry differences between the two Apples in his book ***Understanding the Apple IIe****, published by Brady Communications of Bowie, MD. On pages 8-16 through 8-19 he describes them—they primarily involve video scanning and video signal generation—in great detail.*

According to Sather, "If not for television system incompatibility, the Apple IIe could be made to operate in any country by installing a power supply that would operate from the line voltage of that

country. Supporting the special text requirements of the various languages is no problem because you can simply plug in a keyboard ROM and video ROM for any language....The PAL circuitry is nearly identical to the circuitry of the Apple II Eurocolor card, so basically an Apple IIe PAL motherboard is an American motherboard with a 14.25 MHz oscillator (14.31818 MHz is standard on NTSC Apple IIs), a 50 Hz IOU (60 Hz on NTSC), foreign language video and keyboard ROMS, and a built-in Eurocolor card."

One factor Sather doesn't explain in detail is the difference in the component and slot layouts between the two motherboards. The two are not the same. Devices that plug into the motherboard of one flavor of IIe or that use jumpers to motherboard components are unlikely to work with the other flavor. One problem frequently mentioned in ***Apple User*** *(the only European Apple magazine published in a language I can read) is that slot 3 and the auxiliary slot lie end-to-end on the PAL motherboard, consequently they can't both be used at the same time.*

RAM found in accelerators

E.J. Martin's letter in your November issue (page 2.76) brings up an interesting point about accelerator cards.

These cards use fast-access RAM (150ns or faster) chips that can be accessed at 3 MHz plus, rather than at the standard 1.023 MHz that the motherboard RAM can handle. The 80K accelerator Martin mentions has 48K to replace the motherboard memory, 16K for the language card, and 12K for a copy of what's in the motherboard ROMs. 4K is wasted simply because it's easier and less expensive to put 80K of RAM on the board than 76K.

However, Applied Engineering's TransWarp accelerator takes a different approach. It has 256K of RAM split between the 76K mentioned above and 64K used to accelerate Apple IIe auxiliary memory. This allows programs running in auxiliary memory to be accelerated. What isn't documented is that some of the extra memory can be accessed as a RAM card.

On an old Apple II or on a II-Plus the 64K of auxiliary memory can be used! There are a couple of limitations—neither 80-columns nor double hi-res are added, since that circuitry doesn't exist on these machines. In addition, most programs, including ProDOS, do not realize that the memory is there, since the signature bytes indicate that the machine is not an Apple IIe. Nevertheless, with programs that check to see if the memory is available, or with programs that can be fooled into thinking they are running on a Apple IIe, that memory can be accessed. For example, try a DOS 3.3 RAMdrive program that is designed to work with an extended 80-column card. Many will work on a II-Plus with a TransWarp.

From the other point of view, if you plug a TransWarp into an Apple IIe, then it will still have the 16K RAM card normally used for the II-Plus mode. Many programs will access extra 16K RAM cards. If you tell such programs that you have a 16K card installed in the TransWarp slot it will find and use it.

Philip Chien
Earth News
Titusville, Fla.

Some benchmarks

I am considering upgrading to a IIgs, but I have some questions that I hope you can answer. First of all, I was originally considering a Macintosh, but since the IIgs has arrived I can't make up my mind because of compatibility issues. Compatibility with the IIe and IIc is no problem, but I'm interested in Mac compatibility. I know that the Macintosh and the IIgs are not software compatible, only hardware compatible. However, how identical is the IIgs 128K ROM with the 128K Mac ROM? Specifically, are the QuickDraw, math and other routines 100 per cent compatible on a functional level? As an example, can a Macintosh MicroSoft Basic program that calls QuickDraw routines be simply downloaded from a BBS and run on a IIgs?

By the way, have you run any benchmarks comparing the IIgs to the IIe or Mac Plus? Speed was one of the reasons why I considered dumping my IIe system for a Macintosh Plus, or even (gasp!) an IBM PC.

J. M. Maing
Honolulu, Hawaii

Although the user interface shown in most IIgs demos is similar to the Macintosh, there is almost no other area of compatibility. The peripheral interface ports (disk ports, serial ports, desktop bus for the keyboard) are similar to what the Macintosh has (or is about to get) but only because Apple wants the two computers to use a common family of peripherals. Internally, the IIgs and Mac are nowhere near the same.

This includes "ROM compatibility". The design philosophy of the IIgs ROMs is similar to the Mac's, but the actual routines are not the same. After all, the computers use different processors (68000 series for the Mac, 65816 for the IIgs) and the hardware that must be manipulated by the ROM-based firmware is different.

The ROM routines are accessed as "tools" through a single entry point in each machine's ROM. Many of the tools have the same names and functions, but the exact nature of each call isn't necessarily the same because of the differences between the machines. The IIgs version of QuickDraw (QuickDraw II), for example, is scaled for the pixel dimensions of the IIgs display and must support color.

Apple's engineers are working on some utilities to allow translating things such as graphics, fonts, and files between the two machines, but I will be surprised if we ever see any kind of compatibility at a program level.

The folks who developed Apple's SANE packages say that, based on a sieve of Eratosthenes benchmark involving SANE on the IIgs and Mac, the IIgs runs at about half the speed of the Mac. Since the 65816 has no multiply/divide instructions like the 68000, the SANE benchmark may be a worst-case (for the IIgs) comparison between the two machines.

The Apple II family shines, on the other hand, in the standard benchmark tests used by ***Byte*** *magazine (see the June 1984 issue, page 327, and the October 1984 issue, page 33). Dennis and I ran* ***Byte's*** *benchmarks on a IIe, a IIgs, and a IIe with a TransWarp accelerator and found the latter two to be comparable to an IBM-PC/AT or to a Macintosh-Plus. As with all benchmarks, use care in forming opinions based on these numbers—in particular, they may say more about the Basics used than about the machines themselves (Applesoft on the II family, BASICA on the IBM, Microsoft Basic 1.0 on the Macintosh). On the Apple IIs, the disk read and write tests didn't change with computer speed, but the type of disk drive used did make a big difference, so we've reported those numbers that way. ProDOS was used for the disk tests.*

The comparison data for the IBM PC and PC/AT were taken from ***Byte****, May 1985, page 274; for the Macintosh and Macintosh Plus from November 1986, page 248. The benchmark programs use single-precision arithmetic—this means 5-byte precision for Applesoft versus 4-byte precision for the MicroSoft interpreters on the other machines.*

Byte magazine benchmarks, in seconds

	calc	sieve	write	read
IIe	97	245	--	--
IIgs	35	96	--	--
IIe+TransWarp	30	80	--	--
II + 5.25 drive	--	--	37	36
II + Uni 3.5	--	--	36	22
II + RAMdisk	--	--	9	5
IBM-PC	69	191	56	46
IBM-PC AT	27	80	26	24
Macintosh	79	125	15	10
Macintosh-Plus	79	96	25	23

As you can see, with these benchmarks the IIgs and the accelerated IIe compare very favorably with the 6 MHz IBM PC/AT (for a lot less money) and blow away the others, though the Macintosh appears to have faster drives. The tests don't take into account the use of a math coprocessor for the IBM computers, which would give them a decided advantage if the language the benchmark was written in used the coprocessor (the standard interpreted Microsoft Basic used for these tests does not). On the other hand, a Basic written to take full advantage of 65816 chip on the IIgs (there isn't one at the moment) should execute faster than Applesoft.

Open-Apple

[redacted]

is written, edited, published, and

Business Consultant Richard Barger
Technical Consultant Dennis Doms
Circulation Manager Sally Tally

Open-Apple
P.O. Box 7651
Overland Park, Kansas 66207 U.S.A.

ISSN 0885-4017. Published monthly since January 1985. World-wide prices (in U.S. dollars: airmail delivery included at no additional charge): $24 for 1 year; $44 for 2 years; $60 for 3 years. All back issues are currently available for $2 each; a bound, indexed edition of Volume 1 is $14.95. Index mailed with the February issue. Please send all subscription-related correspondence to:

Open-Apple
P.O. Box 6331
Syracuse, N.Y. 13217 U.S.A.

Subscribers in Australia and New Zealand should send subscription correspondence to ***Open-Apple***, c/o Cybernetic Research Ltd, 576 Malvern Road, Prahran, Vic. 3181, AUSTRALIA.

Open-Apple is available on disk for speech synthesizer users from Speech Enterprises, P.O. Box 7986, Houston, Texas 77270 (713-461-1666)

Unlike most commerical software, ***Open-Apple*** is sold in an unprotected format for your convenience. You are encourgaged to make back-up archival copies or easy-to-read enlarged copies for your own use without charge. You may also copy ***Open-Apple*** for distribution to others. The distribution fee is 15 cents per page per copy distributed.

Source Mail: TCF238 CompuServe: 70120,202